CS 4414 Lab 3—Creating a User-Level Thread Package
Part I
Due: 10pm Wed, Feb 28

1. Project Description

In this project, you will learn about creating and context-switching among concurrent threads sharing an address space. UNIX classically provided the notion of a "process" for an activity and switches between processes to support concurrent activities. The address space (code, data and stack) together with the registers and other state information constitute the state of a process, and these have to be saved and restored when the operating system switches from one process to another. In this project, the unit of activity is a "thread" instead of a process. A thread adds an extra set of CPU state (PC, registers, etc.) and a stack, but shares the rest of the address space with other threads.

You will have a single UNIX process, and you will need to create and manage multiple threads within this UNIX process and address space—in other words, you will implement "user-level" or "many to one" threads. The UNIX kernel will still see this as one single process, and it is up to the library you develop to provide the thread creation/termination/management routines to the user program. The interface routines of the library to be implemented are described below. Use the information given in this handout as a higher level description of what you are expected to implement. Sample programs to assist in testing your implementation are available as additional attachments for this assignment in Collab.

You should be able to switch from one thread to another on an explicit `uthread_yield()` call which a thread can invoke. Remember that, unlike processes, all the threads of a process share the same code and data. Therefore, you do not have to switch the code and data when a thread yields. Only the stack pointer, registers etc. need to be saved and restored. In addition, you should implement a form of round-robin scheduling scheme to switch among runnable threads upon a yield. Note that in this first stage of implementation, we are only supporting non-preemptive scheduling. In the second part of this assignment, we will add pre-emptive scheduling, priorities, and support for synchronization among threads.

Note: You *must* use the make/swapcontext() family of calls (or some other routines that directly manipulate contexts, if you prefer, but this is not recommended). You *may not* use pthreads or any other threading library.

2. Library Interface

Your assignment will be implemented in Linux using C. The API calls you must support are:

• `void uthread_init()`

Initialize the thread library.

• `int uthread_create(void (*func)(int),int val,int pri)`

Create a new thread with priority `pri` that starts execution with function `func` whose argument is `val`; i.e., the prototype of `func` is `void func(int val)`. The uthread_create() function returns the identifier of the thread thus created. For now, the "pri" filed is unused. When creating a thread, please allocate a stack of 1 MByte. The created thread should not run immediately; it should be put at the end of the queue of threads waiting to run.

if a thread returns from the "func" specified in uthread_create, then it should exit.

• `void uthread_yield()`

This is for the currently running thread to relinquish the CPU. Remember that we are only supporting non-preemptive scheduling so far, but when the current thread yields, if you have multiple threads suspended due to their having yielded, you should switch to the one that has been suspended the longest (i.e., round-robin).

• `void uthread_exit()`

Terminate the currently running thread and run the next thread, if there is one. If the exiting thread is the only thread in the process, the entire process should exit. Also note that when the system call exit() is called for any reason, the entire process, with all its threads, should exit. Calling uthread_exit() is the *only* way a thread should exit (unless it is using exit() and terminating the entire process).

Assignment adapted from Prof. Sudhanva Gurumurthi

We will test your library with various test programs that call your thread package via the above API. We will link our test programs against your .o file implementing this API.

Threads should not exit without calling uthread_exit(). This means they should not return from the "func" specified in uthread_create. If they do return from "func" without calling uthread_exit(), you may handle this case any way you choose, except that the program should not crash.

If any thread calls the system call exit(), the whole process should terminate.

3. Useful UNIX System Calls

• **To create a new thread:** `getcontext(), then makecontext()`

(The definition of the `ucontext_t` structure can be found in `/usr/include/sys/ucontext.h`)

• **To switch between threads:** `setcontext()` or `swapcontext()`

(See the man pages)

4. What to Turn in

Remember: we'd rather see something basic that works rather than something sophisticated that doesn't work at all. Get the most rudimentary functionality working first and build from there.

a. A report (at most 5 pages) with a detailed description of how each of the functionalities above was implemented, the design decisions you made, and an explanation of why you made those decisions. Also please note how many hours you spent on the implementation (not counting writing the report). PDF format is preferred.

b. Please upload a single gzipped tar file on Collab containing your report, code, and your PARTNERS file, which includes the pledge and both partners' name. You should also include a separate text file named CONTRIBUTIONS that briefly describes the contributions of each team member.

The code should build "threadpkg1.o" from a Makefile at the command line in the VM environment when we enter "make threadpkg1".

Assignment adapted from Prof. Sudhanva Gurumurthi